

EV316935199

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

APPLICATION FOR LETTERS PATENT

Mechanism for Analyzing Partially Unresolved Input

Inventor(s):

Jeffrey P. Snover

James W. Truher III

Kaushik Pushpavanam

ATTORNEY'S DOCKET NO. MS1-1741US

TECHNICAL FIELD

Subject matter disclosed herein relates to operating environments, and in particular to a mechanism for resolving input entered within an operating environment.

BACKGROUND OF THE INVENTION

Information processed within a computer is classified into one of several data types, such as an integer, a floating point number, a character, and the like. Traditionally, programmers were required to declare the data type of every data object before compilation. During compilation, symbolic addresses were assigned. These symbolic addresses were later replaced with computer addresses for runtime operation. Because the data type had to be known at compile time, these traditional operating environments were referred to as tightly bound environments.

Today, operating environments allow code to discover information about a data type during runtime by “reflecting” on an object. Reflection allows an application to query object metadata to discover information about the object, such as properties, method, fields, and the like. The operating environments support a fixed set of metadata for each object class.

While these reflection-based operating environments provide greater flexibility (e.g., binding to and calling methods at runtime) than traditional tightly bound operating environments, there is still a need for an operating environment that supports greater flexibility with resolving newly created objects and already existing objects.

SUMMARY OF THE INVENTION

The present mechanism provides various capabilities for resolving strings within a command string. The present mechanism operates within an interactive operating environment by receiving a plurality of strings. For any string this is only partially resolved, the mechanism initiates analysis for completely resolving the string. The mechanisms support wildcarding, property sets, relations, conversions, property paths, extended types, data type coercing, and the like.

The mechanism allows third party developers to create new data types and incorporate them into the operating environment. In addition, the mechanism supports a robust grammar on the command line for navigating within an object.

BRIEF DESCRIPTION OF THE DRAWINGS

FIGURE 1 illustrates an exemplary computing device that may use an exemplary administrative tool environment.

FIGURE 2 is a block diagram generally illustrating an overview of an exemplary administrative tool framework for the present administrative tool environment.

FIGURE 3 is a block diagram illustrating components within the host-specific components of the administrative tool framework shown in FIGURE 2.

FIGURE 4 is a block diagram illustrating components within the core engine component of the administrative tool framework shown in FIGURE 2.

FIGURE 5 is one exemplary data structure for specifying a cmdlet suitable for use within the administrative tool framework shown in FIGURE 2.

FIGURE 6 is an exemplary data structure for specifying a command base type from which a cmdlet shown in FIGURE 5 is derived.

1 FIGURE 7 is another exemplary data structure for specifying a cmdlet
2 suitable for use within the administrative tool framework shown in FIGURE 2.

3 FIGURE 8 is a logical flow diagram illustrating an exemplary process for
4 host processing that is performed within the administrative tool framework shown
5 in FIGURE 2.

6 FIGURE 9 is a logical flow diagram illustrating an exemplary process for
7 handling input that is performed within the administrative tool framework shown
8 in FIGURE 2.

9 FIGURE 10 is a logical flow diagram illustrating a process for processing
10 scripts suitable for use within the process for handling input shown in FIGURE 9.

11 FIGURE 11 is a logical flow diagram illustrating a script pre-processing
12 process suitable for use within the script processing process shown in FIGURE 10.

13 FIGURE 12 is a logical flow diagram illustrating a process for applying
14 constraints suitable for use within the script processing process shown in FIGURE
15 10.

16 FIGURE 13 is a functional flow diagram illustrating the processing of a
17 command string in the administrative tool framework shown in FIGURE 2.

18 FIGURE 14 is a logical flow diagram illustrating a process for processing
19 commands strings suitable for use within the process for handling input shown in
20 FIGURE 9.

21 FIGURE 15 is a logical flow diagram illustrating an exemplary process for
22 creating an instance of a cmdlet suitable for use within the processing of command
23 strings shown in FIGURE 14.

1 FIGURE 16 is a logical flow diagram illustrating an exemplary process for
2 populating properties of a cmdlet suitable for use within the processing of
3 commands shown in FIGURE 14.

4 FIGURE 17 is a logical flow diagram illustrating an exemplary process for
5 executing the cmdlet suitable for use within the processing of commands shown in
6 FIGURE 14.

7 FIGURE 18 is a functional block diagram of an exemplary extended type
8 manager suitable for use within the administrative tool framework shown in
9 FIGURE 2.

10 FIGURE 19 graphically depicts exemplary sequences for output processing
11 cmdlets within a pipeline.

12 FIGURE 20 illustrates exemplary processing performed by one of the
13 output processing cmdlets shown in FIGURE 19.

14 FIGURE 21 graphically depicts an exemplary structure for display
15 information accessed during the processing of FIGURE 20.

16 FIGURE 22 is a table listing an exemplary syntax for exemplary output
17 processing cmdlets.

18 FIGURE 23 illustrates results rendered by the out/console cmdlet using
19 various pipeline sequences of the output processing cmdlets.

20 **DETAILED DESCRIPTION**

21 Briefly stated, the present mechanism provides various capabilities for
22 resolving strings within a command string. The present mechanism operates
23 within an interactive operating environment by receiving a plurality of strings.
24 For any string this is partially resolved, the mechanism initiates analysis for
25 completely resolving the string. The mechanisms support wildcarding, property

1 sets, relations, conversions, property paths, extended types, data type coercing, and
2 the like.

3 The following description sets forth a specific exemplary administrative
4 tool environment in which the mechanism operates. Other exemplary
5 environments may include features of this specific embodiment and/or other
6 features, which aim to extend data types and resolve command strings.

7 The following detailed description is divided into several sections. A first
8 section describes an illustrative computing environment in which the
9 administrative tool environment may operate. A second section describes an
10 exemplary framework for the administrative tool environment. Subsequent
11 sections describe individual components of the exemplary framework and the
12 operation of these components. For example, the section on “Exemplary Extended
13 Type Manager”, in conjunction with FIGURE 18, describes an exemplary
14 mechanism for extending the data types available with operating environments and
15 for analyzing partially resolved input.

16 Exemplary Computing Environment

17 FIGURE 1 illustrates an exemplary computing device that may be used in
18 an exemplary administrative tool environment. In a very basic configuration,
19 computing device typically includes at least one processing unit and system
20 memory 104. Depending on the exact configuration and type of computing
21 device, system memory may be volatile (such as RAM), non-volatile (such as
22 ROM, flash memory, etc.) or some combination of the two. System
23 memory typically includes an operating system 105, one or more program
24 modules 106, and may include program data 107. The operating system include a
25 component-based framework that supports components (including properties and

1 events), objects, inheritance, polymorphism, reflection, and provides an object-
2 oriented component-based application programming interface (API), such as that
3 of the .NETTM Framework manufactured by Microsoft Corporation, Redmond,
4 WA. The operating system also includes an administrative tool framework that
5 interacts with the component-based framework to support development of
6 administrative tools (not shown). This basic configuration is illustrated in
7 FIGURE 1 by those components within dashed line 108.

8 Computing device may have additional features or functionality. For
9 example, computing device may also include additional data storage devices
10 (removable and/or non-removable) such as, for example, magnetic disks, optical
11 disks, or tape. Such additional storage is illustrated in FIGURE 1 by removable
12 storage and non-removable storage 110. Computer storage media may include
13 volatile and nonvolatile, removable and non-removable media implemented in any
14 method or technology for storage of information, such as computer readable
15 instructions, data structures, program modules, or other data. System memory
16 104, removable storage and non-removable storage are all examples of computer
17 storage media. Computer storage media includes, but is not limited to, RAM,
18 ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital
19 versatile disks (DVD) or other optical storage, magnetic cassettes, magnetic tape,
20 magnetic disk storage or other magnetic storage devices, or any other medium
21 which can be used to store the desired information and which can be accessed by
22 computing device 100. Any such computer storage media may be part of device
23 100. Computing device may also have input device(s) such as keyboard, mouse,
24 pen, voice input device, touch input device, etc. Output device(s) such as a
25

1 display, speakers, printer, etc. may also be included. These devices are well know
2 in the art and need not be discussed at length here.

3 Computing device may also contain communication connections that allow
4 the device to communicate with other computing devices 118, such as over a
5 network. Communication connections are one example of communication media.
6 Communication media may typically be embodied by computer readable
7 instructions, data structures, program modules, or other data in a modulated data
8 signal, such as a carrier wave or other transport mechanism, and includes any
9 information delivery media. The term “modulated data signal” means a signal that
10 has one or more of its characteristics set or changed in such a manner as to encode
11 information in the signal. By way of example, and not limitation, communication
12 media includes wired media such as a wired network or direct-wired connection,
13 and wireless media such as acoustic, RF, infrared and other wireless media. The
14 term computer readable media as used herein includes both storage media and
15 communication media.

16 Exemplary Administrative Tool Framework

17 FIGURE 2 is a block diagram generally illustrating an overview of an
18 exemplary administrative tool framework 200. Administrative tool
19 framework includes one or more host components 202, host-specific components
20 204, host-independent components 206, and handler components 208. The host-
21 independent components may communicate with each of the other components
22 (i.e., the host components 202, the host-specific components 204, and the handler
23 components 208). Each of these components are briefly described below and
24 described in further detail, as needed, in subsequent sections.
25

Host components

The host components include one or more host programs (e.g., host programs 210-214) that expose automation features for an associated application to users or to other programs. Each host program 210-214 may expose these automation features in its own particular style, such as via a command line, a graphical user interface (GUI), a voice recognition interface, application programming interface (API), a scripting language, a web service, and the like. However, each of the host programs 210-214 expose the one or more automation features through a mechanism provided by the administrative tool framework.

In this example, the mechanism uses cmdlets to surface the administrative tool capabilities to a user of the associated host program 210-214. In addition, the mechanism uses a set of interfaces made available by the host to embed the administrative tool environment within the application associated with the corresponding host program 210-214. Throughout the following discussion, the term “cmdlet” is used to refer to commands that are used within the exemplary administrative tool environment described with reference to FIGURES 2-23.

Cmdlets correspond to commands in traditional administrative environments. However, cmdlets are quite different than these traditional commands. For example, cmdlets are typically smaller in size than their counterpart commands because the cmdlets can utilize common functions provided by the administrative tool framework, such as parsing, data validation, error reporting, and the like. Because such common functions can be implemented once and tested once, the use of cmdlets throughout the administrative tool framework allows the incremental development and test costs associated with

1 application-specific functions to be quite low compared to traditional
2 environments.

3 In addition, in contrast to traditional environments, cmdlets do not need to
4 be stand-alone executable programs. Rather, cmdlets may run in the same
5 processes within the administrative tool framework. This allows cmdlets to
6 exchange “live” objects between each other. This ability to exchange “live”
7 objects allows the cmdlets to directly invoke methods on these objects. The
8 details for creating and using cmdlets are described in further detail below.

9 In overview, each host program **210-214** manages the interactions between
10 the user and the other components within the administrative tool framework.
11 These interactions may include prompts for parameters, reports of errors, and the
12 like. Typically, each host program **210-213** may provide its own set of specific
13 host cmdlets (e.g., host cmdlets **218**). For example, if the host program is an email
14 program, the host program may provide host cmdlets that interact with mailboxes
15 and messages. Even though FIGURE 2 illustrates host programs **210-214**, one
16 skilled in the art will appreciate that host components may include other host
17 programs associated with existing or newly created applications. These other host
18 programs will also embed the functionality provided by the administrative tool
19 environment within their associated application. The processing provided by a
20 host program is described in detail below in conjunction with FIGURE 8.

21 In the examples illustrated in FIGURE 2, a host program may be a
22 management console (i.e., host program **210**) that provides a simple, consistent,
23 administration user interface for users to create, save, and open administrative
24 tools that manage the hardware, software, and network components of the
25

1 computing device. To accomplish these functions, host program provides a set of
2 services for building management GUIs on top of the administrative tool
3 framework. The GUI interactions may also be exposed as user-visible scripts that
4 help teach the users the scripting capabilities provided by the administrative tool
5 environment.

6 In another example, the host program may be a command line interactive
7 shell (i.e., host program **212**). The command line interactive shell may allow shell
8 metadata to be input on the command line to affect processing of the command
9 line.

10 In still another example, the host program may be a web service (i.e., host
11 program **214**) that uses industry standard specifications for distributed computing
12 and interoperability across platforms, programming languages, and applications.

13 In addition to these examples, third parties may add their own host
14 components by creating “third party” or “provider” interfaces and provider
15 cmdlets that are used with their host program or other host programs. The
16 provider interface exposes an application or infrastructure so that the application
17 or infrastructure can be manipulated by the administrative tool framework. The
18 provider cmdlets provide automation for navigation, diagnostics, configuration,
19 lifecycle, operations, and the like. The provider cmdlets exhibit polymorphic
20 cmdlet behavior on a completely heterogeneous set of data stores. The
21 administrative tool environment operates on the provider cmdlets with the same
22 priority as other cmdlet classes. The provider cmdlet is created using the same
23 mechanisms as the other cmdlets. The provider cmdlets expose specific
24 functionality of an application or an infrastructure to the administrative tool
25

1 framework. Thus, through the use of cmdlets, product developers need only create
2 one host component that will then allow their product to operate with many
3 administrative tools. For example, with the exemplary administrative tool
4 environment, system level graphical user interface help menus may be integrated
5 and ported to existing applications.

6 **Host-specific components**

7 The host-specific components include a collection of services that
8 computing systems (e.g., computing device in FIGURE 1) use to isolate the
9 administrative tool framework from the specifics of the platform on which the
10 framework is running. Thus, there is a set of host-specific components for each
11 type of platform. The host-specific components allow the users to use the same
12 administrative tools on different operating systems.

13 Turning briefly to FIGURE 3, the host-specific components may include an
14 intellisense/metadata access component 302, a help cmdlet component 304, a
15 configuration/registration component 306, a cmdlet setup component 308, and an
16 output interface component 309. Components 302-308 communicate with a
17 database store manager associated with a database store 314. The parser and script
18 engine communicate with the intellisense/metadata access component 302. The
19 core engine communicates with the help cmdlet component 304, the
20 configuration/registration component 306, the cmdlet setup component 308, and
21 the output interface component 309. The output interface component 309 includes
22 interfaces provided by the host to out cmdlets. These out cmdlets can then call the
23 host's output object to perform the rendering. Host-specific components may also
24 include a logging/auditing component 310, which the core engine 224 uses to
25

1 communicate with host specific (i.e., platform specific) services that provide
2 logging and auditing capabilities.

3 In one exemplary administrative tool framework, the intellisense/metadata
4 access component provides auto-completion of commands, parameters, and
5 parameter values. The help cmdlet component provides a customized help system
6 based on a host user interface.

7 **Handler components**

8 Referring back to FIGURE 2, the handler component includes legacy
9 utilities 230, management cmdlets 232, non-management cmdlets 234, remoting
10 cmdlets 236, and a web service interface 238. The management cmdlets (also
11 referred to as platform cmdlets) include cmdlets that query or manipulate the
12 configuration information associated with the computing device. Because
13 management cmdlets manipulate system type information, they are dependant upon
14 a particular platform. However, each platform typically has management
15 cmdlets that provide similar actions as management cmdlets on other platforms. For
16 example, each platform supports management cmdlets that get and set system
17 administrative attributes (e.g., get/process, set/IPAddress). The host-independent
18 components communicate with the management cmdlets via cmdlet objects
19 generated within the host-independent components 206. Exemplary data
20 structures for cmdlets objects will be described in detail below in conjunction with
21 FIGURES 5-7.

22 The non-management cmdlets (sometimes referred to as base cmdlets)
23 include cmdlets that group, sort, filter, and perform other processing on objects
24 provided by the management cmdlets 232. The non-management cmdlets 234
25

1 may also include cmdlets for formatting and outputting data associated with the
2 pipelined objects. An exemplary mechanism for providing a data driven command
3 line output is described below in conjunction with FIGURES 19-23. The non-
4 management cmdlets may be the same on each platform and provide a set of
5 utilities that interact with host-independent components via cmdlet objects. The
6 interactions between the non-management cmdlets and the host-independent
7 components allow reflection on objects and allow processing on the reflected
8 objects independent of their (object) type. Thus, these utilities allow developers to
9 write non-management cmdlets once and then apply these non-management
10 cmdlets across all classes of objects supported on a computing system. In the past,
11 developers had to first comprehend the format of the data that was to be processed
12 and then write the application to process only that data. As a consequence,
13 traditional applications could only process data of a very limited scope. One
14 exemplary mechanism for processing objects independent of their object type is
15 described below in conjunction with FIGURE 18.

16 The legacy utilities include existing executables, such as win32 executables
17 that run under cmd.exe. Each legacy utility communicates with the administrative
18 tool framework using text streams (i.e., stdin and stdout), which are a type of
19 object within the object framework. Because the legacy utilities utilize text
20 streams, reflection-based operations provided by the administrative tool
21 framework are not available. The legacy utilities execute in a different process
22 than the administrative tool framework. Although not shown, other cmdlets may
23 also operate out of process.
24
25

1 The remoting cmdlets **236**, in combination with the web service interface
2 **238**, provide remoting mechanisms to access interactive and programmatic
3 administrative tool environments on other computing devices over a
4 communication media, such as internet or intranet (e.g., internet/intranet shown in
5 FIGURE 2). In one exemplary administrative tool framework, the remoting
6 mechanisms support federated services that depend on infrastructure that spans
7 multiple independent control domains. The remoting mechanism allows scripts to
8 execute on remote computing devices. The scripts may be run on a single or on
9 multiple remote systems. The results of the scripts may be processed as each
10 individual script completes or the results may be aggregated and processed en-
11 masse after all the scripts on the various computing devices have completed.

12 For example, web services shown as one of the host components **202** may be
13 a remote agent. The remote agent handles the submission of remote command
14 requests to the parser and administrative tool framework on the target system. The
15 remoting cmdlets serve as the remote client to provide access to the remote agent.
16 The remote agent and the remoting cmdlets communicate via a parsed stream.
17 This parsed stream may be protected at the protocol layer, or additional cmdlets
18 may be used to encrypt and then decrypt the parsed stream.

19 **Host-independent components**

20 The host-independent components include a parser **220**, a script engine and a
21 core engine **224**. The host-independent components provide mechanisms and
22 services to group multiple cmdlets, coordinate the operation of the cmdlets, and
23 coordinate the interaction of other resources, sessions, and jobs with the cmdlets.

24 **Exemplary Parser**

25

1 The parser provides mechanisms for receiving input requests from various
2 host programs and mapping the input requests to uniform cmdlet objects that are
3 used throughout the administrative tool framework, such as within the core engine
4 **224.** In addition, the parser may perform data processing based on the input
5 received. One exemplary method for performing data processing based on the
6 input is described below in conjunction with FIGURE 12. The parser of the
7 present administrative tool framework provides the capability to easily expose
8 different languages or syntax to users for the same capabilities. For example,
9 because the parser is responsible for interpreting the input requests, a change to the
10 code within the parser that affects the expected input syntax will essentially affect
11 each user of the administrative tool framework. Therefore, system administrators
12 may provide different parsers on different computing devices that support different
13 syntax. However, each user operating with the same parser will experience a
14 consistent syntax for each cmdlet. In contrast, in traditional environments, each
15 command implemented its own syntax. Thus, with thousands of commands, each
16 environment supported several different syntax, usually many of which were
17 inconsistent with each other.

18 **Exemplary Script Engine**

19 The script engine provides mechanisms and services to tie multiple cmdlets
20 together using a script. A script is an aggregation of command lines that share
21 session state under strict rules of inheritance. The multiple command lines within
22 the script may be executed either synchronously or asynchronously, based on the
23 syntax provided in the input request. The script engine has the ability to process
24 control structures, such as loops and conditional clauses and to process variables
25

1 within the script. The script engine also manages session state and gives cmdlets
2 access to session data based on a policy (not shown).

3 **Exemplary Core Engine**

4 The core engine is responsible for processing cmdlets identified by the
5 parser 220. Turning briefly to FIGURE 4, an exemplary core engine within the
6 administrative tool framework is illustrated. The exemplary core engine includes a
7 pipeline processor 402, a loader 404, a metadata processor 406, an error & event
8 handler 408, a session manager 410, and an extended type manager 412.

9 Exemplary Metadata Processor

10 The metadata processor is configured to access and store metadata within a
11 metadata store, such as database store 314 shown in FIGURE 3. The metadata
12 may be supplied via the command line, within a cmdlet class definition, and the
13 like. Different components within the administrative tool framework may request
14 the metadata when performing their processing. For example, parser may request
15 metadata to validate parameters supplied on the command line.

16 Exemplary Error & Event Processor

17 The error & event processor provides an error object to store information
18 about each occurrence of an error during processing of a command line. For
19 additional information about one particular error and event processor which is
20 particularly suited for the present administrative tool framework, refer to U.S.
21 Patent Application No. ____ / U.S. Patent No. ____, entitled "System and
22 Method for Persisting Error Information in a Command Line Environment", which
23 is owned by the same assignee as the present invention, and is incorporated here
24 by reference.
25

Exemplary Session Manager

The session manager supplies session and state information to other components within the administrative tool framework 200. The state information managed by the session manager may be accessed by any cmdlet, host, or core engine via programming interfaces. These programming interfaces allow for the creation, modification, and deletion of state information.

Exemplary Pipeline Processor and Loader

The loader is configured to load each cmdlet in memory in order for the pipeline processor to execute the cmdlet. The pipeline processor includes a cmdlet processor and a cmdlet manager 422. The cmdlet processor dispatches individual cmdlets. If the cmdlet requires execution on a remote, or a set of remote machines, the cmdlet processor coordinates the execution with the remoting cmdlet shown in FIGURE 2. The cmdlet manager handles the execution of aggregations of cmdlets. The cmdlet manager 422, the cmdlet processor 420, and the script engine (FIGURE 2) communicate with each other in order to perform the processing on the input received from the host program 210-214. The communication may be recursive in nature. For example, if the host program provides a script, the script may invoke the cmdlet manager to execute a cmdlet, which itself may be a script. The script may then be executed by the script engine 222. One exemplary process flow for the core engine is described in detail below in conjunction with FIGURE 14.

Exemplary Extended Type Manager

As mentioned above, the administrative tool framework provides a set of utilities that allows reflection on objects and allows processing on the reflected objects independent of their (object) type. The administrative tool

1 frameworkinteracts with the component framework on the computing system
2 (component frameworkin FIGURE 1) to perform this reflection. As one skilled in
3 the art will appreciate, reflection provides the ability to query an object and to
4 obtain a type for the object, and then reflect on various objects and properties
5 associated with that type of object to obtain other objects and/or a desired value.

6 Even though reflection provides the administrative tool frameworka
7 considerable amount of information on objects, the inventors appreciated that
8 reflection focuses on the type of object. For example, when a database datatable is
9 reflected upon, the information that is returned is that the datatable has two
10 properties: a column property and a row property. These two properties do not
11 provide sufficient detail regarding the “objects” within the datatable. Similar
12 problems arise when reflection is used on extensible markup language (XML) and
13 other objects.

14 Thus, the inventors conceived of an extended type managerthat focuses on
15 the usage of the type. For this extended type manager, the type of object is not
16 important. Instead, the extended type manager is interested in whether the object
17 can be used to obtain required information. Continuing with the above datatable
18 example, the inventors appreciated that knowing that the datatable has a column
19 property and a row property is not particularly interesting, but appreciated that one
20 column contained information of interest. Focusing on the usage, one could
21 associate each row with an “object” and associate each column with a “property”
22 of that “object”. Thus, the extended type manager 412 provides a mechanism to
23 create “objects” from any type of precisely parse-able input. In so doing, the
24 extended type managersupplements the reflection capabilities provided by the
25

1 component-based framework 120 and extends “reflection” to any type of precisely
2 parse-able input.

3 In overview, the extended type manager is configured to access precisely
4 parse-able input (not shown) and to correlate the precisely parse-able input with a
5 requested data type. The extended type manager 412 then provides the requested
6 information to the requesting component, such as the pipeline processor 402 or
7 parser 220. In the following discussion, precisely parse-able input is defined as
8 input in which properties and values may be discerned. Some exemplary precisely
9 parse-able input include Windows Management Instrumentation (WMI) input,
10 ActiveX Data Objects (ADO) input, eXtensible Markup Language (XML) input,
11 and object input, such as .NET objects. Other precisely parse-able input may
12 include third party data formats.

13 Turning briefly to FIGURE 18, a functional block diagram of an exemplary
14 extended type manager for use within the administrative tool framework is shown.
15 For explanation purposes, the functionality (denoted by the number “3” within a
16 circle) provided by the extended type manager is contrasted with the functionality
17 provided by a traditional tightly bound system (denoted by the number “1” within
18 a circle) and the functionality provided by a reflection system (denoted by the
19 number “2” within a circle). In the traditional tightly bound system, a caller 1802
20 within an application directly accesses the information (e.g., properties P1 and P2,
21 methods M1 and M2) within object A. As mentioned above, the caller 1802 must
22 know, a priori, the properties (e.g., properties P1 and P2) and methods (e.g.,
23 methods M1 and M2) provided by object A at compile time. In the reflection
24 system, generic code 1820 (not dependent on any data type) queries a system 1808
25

1 that performs reflection 1810 on the requested object and returns the information
2 (e.g., properties P1 and P2, methods M1 and M2) about the object (e.g., object A)
3 to the generic code 1820. Although not shown in object A, the returned
4 information may include additional information, such as vendor, file, date, and the
5 like. Thus, through reflection, the generic code 1820 obtains at least the same
6 information that the tightly bound system provides. The reflection system also
7 allows the caller 1802 to query the system and get additional information without
8 any a priori knowledge of the parameters.

9 In both the tightly bound systems and the reflection systems, new data
10 types can not be easily incorporated within the operating environment. For
11 example, in a tightly bound system, once the operating environment is delivered,
12 the operating environment can not incorporate new data types because it would
13 have to be rebuilt in order to support them. Likewise, in reflection systems, the
14 metadata for each object class is fixed. Thus, incorporating new data types is not
15 usually done.

16 However, with the present extended type manager new data types can be
17 incorporated into the operating system. With the extended type manager 1822,
18 generic code 1820 may reflect on a requested object to obtain extended data types
19 (e.g., object A') provided by various external sources, such as a third party objects
20 (e.g., object A' and B), a semantic web 1832, an ontology service 1834, and the
21 like. As shown, the third party object may extend an existing object (e.g., object
22 A') or may create an entirely new object (e.g., object B).

23 Each of these external sources may register their unique structure within a
24 type metadata 1840 and may provide code 1842. When an object is queried, the
25 extended type manager reviews the type metadata 1840 to determine whether the

1 object has been registered. If the object is not registered within the type metadata
2 1840, reflection is performed. Otherwise, extended reflection is performed. The
3 code 1842 returns the additional properties and methods associated with the type
4 being reflected upon. For example, if the input type is XML, the code 1842 may
5 include a description file that describes the manner in which the XML is used to
6 create the objects from the XML document. Thus, the type metadata 1840
7 describes how the extended type manager 412 should query various types of
8 precisely parse-able input (e.g., third party objects A' and B, semantic web 1832)
9 to obtain the desired properties for creating an object for that specific input type
10 and the code 1842 provides the instructions to obtain these desired properties. As
11 a result, the extended type manager 412 provides a layer of indirection that allows
12 "reflection" on all types of objects.

13 In addition to providing extended types, the extend type manager 412
14 provides additional query mechanisms, such as a property path mechanism, a key
15 mechanism, a compare mechanism, a conversion mechanism, a globber
16 mechanism, a property set mechanism, a relationship mechanism, and the like.
17 Each of these query mechanisms, described below in the section "Exemplary
18 Extended Type Manager Processing", provides flexibility to system administrators
19 when entering command strings. Various techniques may be used to implement
20 the semantics for the extended type manager. Three techniques are described
21 below. However, those skilled in the art will appreciate that variations of these
22 techniques may be used without departing from the scope of the claimed
23 invention.

24 In one technique, a series of classes having static methods (e.g.,
25 getProperty()) may be provided. An object is input into the static method (e.g.,

1 getproperty(object)), and the static method returns a set of results. In another
2 technique, the operating environment envelopes the object with an adapter. Thus,
3 no input is supplied. Each instance of the adapter has a getproperty method that
4 acts upon the enveloped object and returns the properties for the enveloped object.
5 The following is pseudo code illustrating this technique:

6
7 Class Adaptor

8 {
9 Object X;
10 getProperties();
11 }.

12
13 In still another technique, an adaptor class subclasses the object.
14 Traditionally, subclassing occurred before compilation. However, with certain
15 operating environments, subclassing may occur dynamically. For these types of
16 environments, the following is pseudo code illustrating this technique:

17
18 Class Adaptor : A

19 {
20 getProperties()
21 {
22 return data;
23 }
24 }.

1 Thus, as illustrated in FIGURE 18, the extended type manager allows
2 developers to create a new data type, register the data type, and allow other
3 applications and cmdlets to use the new data type. In contrast, in prior
4 administrative environments, each data type had to be known at compile time so
5 that a property or method associated with an object instantiated from that data type
6 could be directly accessed. Therefore, adding new data types that were supported
7 by the administrative environment was seldom done in the past.

8 Referring back to FIGURE 2, in overview, the administrative tool
9 framework does not rely on the shell for coordinating the execution of commands
10 input by users, but rather, splits the functionality into processing portions (e.g.,
11 host-independent components 206) and user interaction portions (e.g., via host
12 cmdlets). In addition, the present administrative tool environment greatly
13 simplifies the programming of administrative tools because the code required for
14 parsing and data validation is no longer included within each command, but is
15 rather provided by components (e.g., parser 220) within the administrative tool
16 framework. The exemplary processing performed within the administrative tool
17 framework is described below.

18 Exemplary Operation

19 FIGURES 5-7 graphically illustrate exemplary data structures used within
20 the administrative tool environment. FIGURES 8-17 graphically illustrate
21 exemplary processing flows within the administrative tool environment. One
22 skilled in the art will appreciate that certain processing may be performed by a
23 different component than the component described below without departing from
24 the scope of the present invention. Before describing the processing performed
25

1 within the components of the administrative tool framework, exemplary data
2 structures used within the administrative tool framework are described.

3 Exemplary Data Structures for Cmdlet Objects

4 FIGURE 5 is an exemplary data structure for specifying a cmdlet suitable
5 for use within the administrative tool framework shown in FIGURE 2. When
6 completed, the cmdlet may be a management cmdlet, a non-management cmdlet, a
7 host cmdlet, a provider cmdlet, or the like. The following discussion describes the
8 creation of a cmdlet with respect to a system administrator's perspective (i.e., a
9 provider cmdlet). However, each type of cmdlet is created in the same manner
10 and operates in a similar manner. A cmdlet may be written in any language, such
11 as C#. In addition, the cmdlet may be written using a scripting language or the
12 like. When the administrative tool environment operates with the .NET
13 Framework, the cmdlet may be a .NET object.

14 The provider cmdlet **500** (hereinafter, referred to as cmdlet **500**) is a public
15 class having a cmdlet class name (e.g., StopProcess **504**). Cmdlet **500** derives
16 from a cmdlet class **506**. An exemplary data structure for a cmdlet class **506** is
17 described below in conjunction with FIGURE 6. Each cmdlet **500** is associated
18 with a command attribute **502** that associates a name (e.g., Stop/Process) with the
19 cmdlet **500**. The name is registered within the administrative tool environment.
20 As will be described below, the parser looks in the cmdlet registry to identify the
21 cmdlet **500** when a command string having the name (e.g., Stop/Process) is
22 supplied as input on a command line or in a script.

23 The cmdlet **500** is associated with a grammar mechanism that defines a
24 grammar for expected input parameters to the cmdlet. The grammar mechanism
25

1 may be directly or indirectly associated with the cmdlet. For example, the cmdlet
2 **500** illustrates a direct grammar association. In this cmdlet **500**, one or more
3 public parameters (e.g., ProcessName **510** and PID **512**) are declared. The
4 declaration of the public parameters drives the parsing of the input objects to the
5 cmdlet **500**. Alternatively, the description of the parameters may appear in an
6 external source, such as an XML document. The description of the parameters in
7 this external source would then drive the parsing of the input objects to the cmdlet.

8 Each public parameter **510**, **512** may have one or more attributes (i.e.,
9 directives) associated with it. The directives may be from any of the following
10 categories: parsing directive **521**, data validation directive **522**, data generation
11 directive **523**, processing directive **524**, encoding directive **525**, and
12 documentation directive **526**. The directives may be surrounded by square
13 brackets. Each directive describes an operation to be performed on the following
14 expected input parameter. Some of the directives may also be applied at a class
15 level, such as user-interaction type directives. The directives are stored in the
16 metadata associated with the cmdlet. The application of these attributes is
17 described below in conjunction with FIGURE 12.

18 These attributes may also affect the population of the parameters declared
19 within the cmdlet. One exemplary process for populating these parameters is
20 described below in conjunction with FIGURE 16. The core engine may apply
21 these directives to ensure compliance. The cmdlet **500** includes a first method **530**
22 (hereinafter, interchangeably referred to as StartProcessing method **530**) and a
23 second method **540** (hereinafter, interchangeably referred to as processRecord
24 method **540**). The core engine uses the first and second methods **530**, **540** to
25

1 direct the processing of the cmdlet **500**. For example, the first method **530** is
2 executed once and performs set-up functions. The code **542** within the second
3 method **540** is executed for each object (e.g., record) that needs to be processed by
4 the cmdlet **500**. The cmdlet **500** may also include a third method (not shown) that
5 cleans up after the cmdlet **500**.

6 Thus, as shown in FIGURE 5, code **542** within the second method **540** is
7 typically quite brief and does not contain functionality required in traditional
8 administrative tool environments, such as parsing code, data validation code, and
9 the like. Thus, system administrators can develop complex administrative tasks
10 without learning a complex programming language.

11 FIGURE 6 is an exemplary data structure **600** for specifying a cmdlet base
12 class **602** from which the cmdlet shown in FIGURE 5 is derived. The cmdlet base
13 class **602** includes instructions that provide additional functionality whenever the
14 cmdlet includes a hook statement and a corresponding switch is input on the
15 command line or in the script (jointly referred to as command input).

16 The exemplary data structure **600** includes parameters, such as Boolean
17 parameter verbose **610**, whatif **620**, and confirm **630**. As will be explained below,
18 these parameters correspond to strings that may be entered on the command input.
19 The exemplary data structure **600** may also include a security method **640** that
20 determines whether the task being requested for execution is allowed.

21 FIGURE 7 is another exemplary data structure **700** for specifying a cmdlet.
22 In overview, the data structure provides a means for clearly expressing a contract
23 between the administrative tool framework and the cmdlet. Similar to data
24 structure **500**, data structure is a public class that derives from a cmdlet class **704**.
25

1 The software developer specifies a cmdletDeclaration 702 that associates a
2 noun/verb pair, such as "get/process" and "format/table", with the cmdlet 700.
3 The noun/verb pair is registered within the administrative tool environment. The
4 verb or the noun may be implicit in the cmdlet name. Also, similar to data
5 structure 500, data structuremay include one or more public members (e.g., Name
6 730, Recurse 732), which may be associated with the one or more directives 520-
7 526 described in conjunction with data structure 500.

8 However, in this exemplary data structure 700, each of the expected input
9 parameters 730 and 732 is associated with an input attribute 731 and 733,
10 respectively. The input attributes 731 and 733 specifying that the data for its
11 respective parameter 730 and 732 should be obtained from the command line.
12 Thus, in this exemplary data structure 700, there are not any expected input
13 parameters that are populated from a pipelined object that has been emitted by
14 another cmdlet. Thus, data structure 700 does not override the first method (e.g.,
15 StartProcessing) or the second method (e.g., ProcessRecord) which are provided
16 by the cmdlet base class.

17 The data structuremay also include a private memberthat is not recognized
18 as an input parameter. The private membermay be used for storing data that is
19 generated based on one of the directives.

20
21 Thus, as illustrated in data structure 700, through the use of declaring
22 public properties and directives within a specific cmdlet class, cmdlet developers
23 can easily specify a grammar for the expected input parameters to their cmdlets
24 and specify processing that should be performed on the expected input parameters
25 without requiring the cmdlet developers to generate any of the underlying logic.

1 Data structure 700 illustrates a direct association between the cmdlet and the
2 grammar mechanism. As mentioned above, this associated may also be indirect,
3 such as by specifying the expected parameter definitions within an external source,
4 such as an XML document.

5 The exemplary process flows within the administrative tool environment
6 are now described.

7 **Exemplary Host Processing Flow**

8 FIGURE 8 is a logical flow diagram illustrating an exemplary process for
9 host processing that is performed within the administrative tool framework shown
10 in FIGURE 2. The process begins at block 801, where a request has been received
11 to initiate the administrative tool environment for a specific application. The
12 request may have been sent locally through keyboard input, such as selecting an
13 application icon, or remotely through the web services interface of a different
14 computing device. For either scenario, processing continues to block 802.

15 At block 802, the specific application (e.g., host program) on the “target”
16 computing device sets up its environment. This includes determining which
17 subsets of cmdlets (e.g., management cmdlets 232, non-management cmdlets 234,
18 and host cmdlets 218) are made available to the user. Typically, the host program
19 will make all the non-management cmdlets 234 available and its own host cmdlets
20 218 available. In addition, the host program will make a subset of the
21 management cmdlets 234 available, such as cmdlets dealing with processes, disk,
22 and the like. Thus, once the host program makes the subsets of cmdlets available,
23 the administrative tool framework is effectively embedded within the
24 corresponding application. Processing continues to block 804.
25

1 At block **804**, input is obtained through the specific application. As
2 mentioned above, input may take several forms, such as command lines, scripts,
3 voice, GUI, and the like. For example, when input is obtained via a command
4 line, the input is retrieve from the keystrokes entered on a keyboard. For a GUI
5 host, a string is composed based on the GUI. Processing continues at block **806**.

6 At block **806**, the input is provided to other components within the
7 administrative tool framework for processing. The host program may forward the
8 input directly to the other components, such as the parser. Alternatively, the host
9 program may forward the input via one of its host cmdlets. The host cmdlet may
10 convert its specific type of input (e.g., voice) into a type of input (e.g., text string,
11 script) that is recognized by the administrative tool framework. For example,
12 voice input may be converted to a script or command line string depending on the
13 content of the voice input. Because each host program is responsible for
14 converting their type of input to an input recognized by the administrative tool
15 framework, the administrative tool framework can accept input from any number
16 of various host components. In addition, the administrative tool framework
17 provides a rich set of utilities that perform conversions between data types when
18 the input is forwarded via one of its cmdlets. Processing performed on the input
19 by the other components is described below in conjunction with several other
20 figures. Host processing continues at decision block **808**.

21 At decision block **808**, a determination is made whether a request was
22 received for additional input. This may occur if one of the other components
23 responsible for processing the input needs additional information from the user in
24 order to complete its processing. For example, a password may be required to
25 access certain data, confirmation of specific actions may be needed, and the like.

1 For certain types of host programs (e.g., voice mail), a request such as this may
2 not be appropriate. Thus, instead of querying the user for additional information,
3 the host program may serialize the state, suspend the state, and send a notification
4 so that at a later time the state may be resumed and the execution of the input be
5 continued. In another variation, the host program may provide a default value
6 after a predetermined time period. If a request for additional input is received,
7 processing loops back to block 804, where the additional input is obtained.
8 Processing then continues through blocks as described above. If no request for
9 additional input is received and the input has been processed, processing continues
10 to block 810.

11 At block 810, results are received from other components within the
12 administrative tool framework. The results may include error messages, status,
13 and the like. The results are in an object form, which is recognized and processed
14 by the host cmdlet within the administrative tool framework. As will be described
15 below, the code written for each host cmdlet is very minimal. Thus, a rich set of
16 output may be displayed without requiring a huge investment in development
17 costs. Processing continues at block 812.

18 At block 812, the results may be viewed. The host cmdlet converts the
19 results to the display style supported by the host program. For example, a returned
20 object may be displayed by a GUI host program using a graphical depiction, such
21 as an icon, barking dog, and the like. The host cmdlet provides a default format
22 and output for the data. The default format and output may utilize the exemplary
23 output processing cmdlets described below in conjunction with FIGURES 19-23.
24 After the results are optionally displayed, the host processing is complete.

25 **Exemplary Process Flows for Handling Input**

FIGURE 9 is a logical flow diagram illustrating an exemplary process for handling input that is performed within the administrative tool framework shown in FIGURE 2. Processing begins at block where input has been entered via a host program and forwarded to other components within the administrative tool framework. Processing continues at block **902**.

At block **902**, the input is received from the host program. In one exemplary administrative tool framework, the input is received by the parser, which deciphers the input and directs the input for further processing. Processing continues at decision block **904**.

At decision block **904**, a determination is made whether the input is a script. The input may take the form of a script or a string representing a command line (hereinafter, referred to as a “command string”). The command string may represent one or more cmdlets pipelined together. Even though the administrative tool framework supports several different hosts, each host provides the input as either a script or a command string for processing. As will be shown below, the interaction between scripts and command strings is recursive in nature. For example, a script may have a line that invokes a cmdlet. The cmdlet itself may be a script.

Thus, at decision block **904**, if the input is in a form of a script, processing continues at block **906**, where processing of the script is performed. Otherwise, processing continues at block **908**, where processing of the command string is performed. Once the processing performed within either block is completed, processing of the input is complete.

Exemplary Processing of Scripts

1 FIGURE 10 is a logical flow diagram illustrating a process for processing a
2 script suitable for use within the process for handling input shown in FIGURE 9.
3 The process begins at block **1001**, where the input has been identified as a script.
4 The script engine and parser communicate with each other to perform the
5 following functions. Processing continues at block **1002**.

6 At block **1002**, pre-processing is performed on the script. Briefly, turning
7 to FIGURE 11, a logical flow diagram is shown that illustrates a script pre-
8 processing process **1100** suitable for use within the script processing process **1000**.
9 Script pre-processing begins at block and continues to decision block **1102**.

10 At decision block **1102**, a determination is made whether the script is being
11 run for the first time. This determination may be based on information obtained
12 from a registry or other storage mechanism. The script is identified from within
13 the storage mechanism and the associated data is reviewed. If the script has not
14 run previously, processing continues at block **1104**.

15 At block **1104**, the script is registered in the registry. This allows
16 information about the script to be stored for later access by components within the
17 administrative tool framework. Processing continues at block **1106**.

18 At block **1106**, help and documentation are extracted from the script and
19 stored in the registry. Again, this information may be later accessed by
20 components within the administrative tool framework. The script is now ready for
21 processing and returns to block **1004** in FIGURE 10.

22 Returning to decision block **1102**, if the process concludes that the script
23 has run previously, processing continues to decision block **1108**. At decision block
24 **1108**, a determination is made whether the script failed during processing. This
25

1 information may be obtained from the registry. If the script has not failed, the
2 script is ready for processing and returns to blockin FIGURE 10.

3 However, if the script has failed, processing continues at block **1110**. At
4 block **1110**, the script engine may notify the user through the host program that the
5 script has previously failed. This notification will allow a user to decide whether
6 to proceed with the script or to exit the script. As mentioned above in conjunction
7 with FIGURE 8, the host program may handle this request in various ways
8 depending on the style of input (e.g., voice, command line). Once additional input
9 is received from the user, the script either returns to blockin FIGURE 10 for
10 processing or the script is aborted.

11 Returning to blockin FIGURE 10, a line from the script is retrieved.
12 Processing continues at decision block **1006**. At decision block **1006**, a
13 determination is made whether the line includes any constraints. A constraint is
14 detected by a predefined begin character (e.g., a bracket “[”) and a corresponding
15 end character (e.g., a close bracket “]”). If the line includes constraints, processing
16 continues to block **1008**.

17 At block **1008**, the constraints included in the line are applied. In general,
18 the constraints provide a mechanism within the administrative tool framework to
19 specify a type for a parameter entered in the script and to specify validation logic
20 which should be performed on the parameter. The constraints are not only
21 applicable to parameters, but are also applicable to any type of construct entered in
22 the script, such as variables. Thus, the constraints provide a mechanism within an
23 interpretive environment to specify a data type and to validate parameters. In
24 traditional environments, system administrators are unable to formally test
25

1 parameters entered within a script. An exemplary process for applying constraints
2 is illustrated in FIGURE 12.

3 At decision block **1010**, a determination is made whether the line from the
4 script includes built-in capabilities. Built-in capabilities are capabilities that are
5 not performed by the core engine. Built-in capabilities may be processed using
6 cmdlets or may be processed using other mechanisms, such as in-line functions. If
7 the line does not have built-in capabilities, processing continues at decision block
8 **1014**. Otherwise, processing continues at block **1012**.

9 At block **1012**, the built-in capabilities provided on the line of the script are
10 processed. Example built-in capabilities may include execution of control
11 structures, such as “if” statements, “for” loops, switches, and the like. Built-in
12 capabilities may also include assignment type statements (e.g., a=3). Once the
13 built-in capabilities have been processed, processing continues to decision block
14 **1014**.

15 At decision block **1014**, a determination is made whether the line of the
16 script includes a command string. The determination is based on whether the data
17 on the line is associated with a command string that has been registered and with a
18 syntax of the potential cmdlet invocation. As mentioned above, the processing of
19 command strings and scripts may be recursive in nature because scripts may
20 include command strings and command strings may execute a cmdlet that is a
21 script itself. If the line does not include a command string, processing continues at
22 decision block **1018**. Otherwise, processing continues at block **1016**.

23 At block **1016**, the command string is processed. In overview, the
24 processing of the command string includes identifying a cmdlet class by the parser
25 and passing the corresponding cmdlet object to the core engine for execution. The

1 command string may also include a pipelined command string that is parsed into
2 several individual cmdlet objects and individually processed by the core engine.
3 One exemplary process for processing command strings is described below in
4 conjunction with FIGURE 14. Once the command string is processed, processing
5 continues at decision block **1018**.

6 At decision block **1018**, a determination is made whether there is another
7 line in the script. If there is another line in the script, processing loops back to
8 block and proceeds as described above in blocks **1004-1016**. Otherwise,
9 processing is complete.

10 An exemplary process for applying constraints in block **1008** is illustrated
11 in FIGURE 12. The process begins at block **1201** where a constraint is detected in
12 the script or in the command string on the command line. When the constraint is
13 within a script, the constraints and the associated construct may occur on the same
14 line or on separate lines. When the constraint is within a command string, the
15 constraint and the associated construct occur before the end of line indicator (e.g.,
16 enter key). Processing continues to block **1202**.

17 At block **1202**, constraints are obtained from the interpretive environment.
18 In one exemplary administrative tool environment, the parser deciphers the input
19 and determines the occurrence of constraints. Constraints may be from one of the
20 following categories: predicate directive, parsing directive, data validation
21 directive, data generation directive, processing directive, encoding directive, and
22 documentation directive. In one exemplary parsing syntax, the directives are
23 surrounded by square brackets and describe the construct that follows them. The
24 construct may be a function, a variable, a script, or the like.
25

1 As will be described below, through the use of directives, script authors are
2 allowed to easily type and perform processing on the parameters within the script
3 or command line (i.e., an interpretive environment) without requiring the script
4 authors to generate any of the underlying logic. Processing continues to block
5 **1204**.

6 At block **1204**, the constraints that are obtained are stored in the metadata
7 for the associated construct. The associated construct is identified as being the
8 first non-attribution token after one or more attribution tokens (tokens that denote
9 constraints) have been encountered. Processing continues to block **1206**.

10 At block **1206**, whenever the construct is encountered within the script or in
11 the command string, the constraints defined within the metadata are applied to the
12 construct. The constraints may include data type, predicate directives **1210**,
13 documentation directives **1212**, parsing directives **1214**, data generation directives
14 **1216**, data validation directives **1218**, and object processing and encoding
15 directives **1220**. Constraints specifying data types may specify any data type
16 supported by the system on which the administrative tool framework is running.
17 Predicate directives **1210** are directives that indicate whether processing should
18 occur. Thus, predicate directives **1210** ensure that the environment is correct for
19 execution. For example, a script may include the following predicate directive:

20 [PredicateScript("isInstalled","ApplicationZ")].
21

22 The predicate directive ensures that the correct application is installed on
23 the computing device before running the script. Typically, system environment
24 variables may be specified as predicate directives. Exemplary directives from
25

1 directive types **1212-1220** are illustrated in Tables 1-5. Processing of the script is
2 then complete.

3 Thus, the present process for applying types and constraints within an
4 interpretive environment, allows system administrators to easily specify a type,
5 specify validation requirements, and the like without having to write the
6 underlying logic for performing this processing. The following is an example of
7 the constraint processing performed on a command string specified as follows:

8 [Integer][ValidationRange(3,5)]\$a=4.

9 There are two constraints specified via attribution tokens denoted by “[]”.
10 The first attribution token indicates that the variable is a type integer and a second
11 attribution token indicates that the value of the variable \$a must be between 3 and
12 5 inclusive. The example command string ensures that if the variable \$a is
13 assigned in a subsequent command string or line, the variable \$a will be checked
14 against the two constraints. Thus, the following command strings would each
15 result in an error:
16

17 \$a = 231

18 \$a = “apple”

19 \$a = \$(get/location).

20 The constraints are applied at various stages within the administrative tool
21 framework. For example, applicability directives, documentation directives, and
22 parsing guideline directives are processed at a very early stage within the parser.
23 Data generation directives and validation directives are processed in the engine
24 once the parser has finished parsing all the input parameters.
25

The following tables illustrate representative directives for the various categories, along with an explanation of the processing performed by the administrative tool environment in response to the directive.

Name	Description
PrerequisiteMachineRoleAttribute	Informs shell whether element is to be used only in certain machine roles (e.g., File Server, Mail Server).
PrerequisiteUserRoleAttribute	Informs shell whether element is to be used only in certain user roles (e.g., Domain Administrator, Backup Operator).
PrerequisiteScriptAttribute	Informs the shell this script will be run before excuting the actual command or parameter. Can be used for parameter validation
PrerequisiteUITypeAttribute	This is used to check the User interface available before excuting

Table 1. Applicability Directives

Name	Description
ParsingParameterPositionAttribute	Maps unqualified

	parameters based on position.
ParsingVariableLengthParameterListAttribute	Maps parameters not having a ParsingParameterPosition attribute.
ParsingDisallowInteractionAttribute	Specifies action when number of parameters is less than required number.
ParsingRequireInteractionAttribute	Specifies that parameters are obtained through interaction.
ParsingHiddenElementAttribute	Makes parameter invisible to end user.
ParsingMandatoryParameterAttribute	Specifies that the parameter is required.
ParsingPasswordParameterAttribute	Requires special handling of parameter.
ParsingPromptStringAttribute	Specifies a prompt for the parameter.
ParsingDefaultAnswerAttribute	Specifies default answer for parameter.
ParsingDefaultAnswerScriptAttribute	Specifies action to

	get default answer for parameter.
ParsingDefaultValueAttribute	Specifies default value for parameter.
ParsingDefaultValueScriptAttribute	Specifies action to get default value for parameter.
ParsingParameterMappingAttribute	Specifies a way to group parameters
ParsingParameterDeclarationAttribute	This defines that the filed is a parameter
ParsingAllowPipelineInputAttribute	Defines the parameter can be populated from the pipeline

Table 2. Parsing Guideline Directives

Name	Description
DocumentNameAttribute	Provides a Name to refer to elements for interaction or help.
DocumentShortDescriptionAttribute	Provides brief description of element.
DocumentLongDescriptionAttribute	Provides detailed description

		of element.
DocumentExampleAttribute		Provides example of element.
DocumentSeeAlsoAttribute		Provides a list of related elements.
DocumentSynopsisAttribute		Provides documentation information for element.

Table 3. Documentation Directives

Name	Description
ValidationRangeAttribute	Specifies that parameter must be within certain range.
ValidationSetAttribute	Specifies that parameter must be within certain collection.
ValidationPatternAttribute	Specifies that parameter must fit a certain pattern.
ValidationLengthAttribute	Specifies the strings must be within size range.
ValidationTypeAttribute	Specifies that parameter must be of certain type.
ValidationCountAttribute	Specifies that input items must be of a certain number.
ValidationFileAttribute	Specifies certain properties for a

	file.
ValidationFileAttributesAttribute	Specifies certain properties for a file.
ValidationFileSizeAttribute	Specifies that files must be within specified range.
ValidationNetworkAttribute	Specifies that given Network Entity supports certain properties.
ValidationScriptAttribute	Specifies conditions to evaluate before using element.
ValidationMethodAttribute	Specifies conditions to evaluate before using element.

Table 4. Data Validation Directives

Name	Description
ProcessingTrimStringAttribute	Specifies size limit for strings.
ProcessingTrimCollectionAttribute	Specifies size limit for collection.
EncodingTypeCoercionAttribute	Specifies Type that objects are to be encoded.
ExpansionWildcardsAttribute	Provides a mechanism to allow globbing

Table 5. Processing and Encoding Directives

When the exemplary administrative tool framework is operating within the .NET™ Framework, each category has a base class that is derived from a basic category class (e.g., CmdAttribute). The basic category class derives from a System.Attribute class. Each category has a pre-defined function (e.g., attrib.func()) that is called by the parser during category processing. The script author may create a custom category that is derived from a custom category class (e.g., CmdCustomAttribute). The script author may also extend an existing category class by deriving a directive class from the base category class for that category and override the pre-defined function with their implementation. The script author may also override directives and add new directives to the pre-defined set of directives.

The order of processing of these directives may be stored in an external data store accessible by the parser. The administrative tool framework looks for registered categories and calls a function (e.g., ProcessCustomDirective) for each of the directives in that category. Thus, the order of category processing may be dynamic by storing the category execution information in a persistent store. At different processing stages, the parser checks in the persistent store to determine if any metadata category needs to be executed at that time. This allows categories to be easily deprecated by removing the category entry from the persistent store.

Exemplary Processing of Command Strings

One exemplary process for processing command strings is now described. FIGURE 13 is a functional flow diagram graphically illustrating the processing of

1 a command string through a parser and a core engine within the administrative tool
2 framework shown in FIGURE 2. The exemplary command string pipelines several
3 commands (i.e., process command 1360, where command 1362, sort command
4 1364, and table command 1366). The command line may pass input parameters to
5 any of the commands (e.g., "handlecount > 400" is passed to the where command
6 1362). One will note that the process command does not have any associated input
7 parameters.

8 In the past, each command was responsible for parsing the input parameters
9 associated with the command, determining whether the input parameters were
10 valid, and issuing error messages if the input parameters were not valid. Because
11 the commands were typically written by various programmers, the syntax for the
12 input parameters on the command line was not very consistent. In addition, if an
13 error occurred, the error message, even for the same error, was not very consistent
14 between the commands.

15 For example, in a UNIX environment, an "ls" command and a "ps"
16 command have many inconsistencies between them. While both accept an option
17 "-w", the "-w" option is used by the "ls" command to denote the width of the page,
18 while the "-w" option is used by the "ps" command to denote print wide output (in
19 essence, ignoring page width). The help pages associated with the "ls" and the
20 "ps" command have several inconsistencies too, such as having options bolded in
21 one and not the other, sorting options alphabetically in one and not the other,
22 requiring some options to have dashes and some not.

23 The present administrative tool framework provides a more consistent
24 approach and minimizes the amount of duplicative code that each developer must
25

1 write. The administrative tool framework provides a syntax (e.g., grammar), a
2 corresponding semantics (e.g., a dictionary), and a reference model to enable
3 developers to easily take advantage of common functionality provided by the
4 administrative tool framework 200.

5 Before describing the present invention any further, definitions for
6 additional terms appearing through-out this specification are provided. Input
7 parameter refers to input-fields for a cmdlet. Argument refers to an input
8 parameter passed to a cmdlet that is the equivalent of a single string in the argv
9 array or passed as a single element in a cmdlet object. As will be described below,
10 a cmdlet provides a mechanism for specifying a grammar. The mechanism may
11 be provided directly or indirectly. An argument is one of an option, an option-
12 argument, or an operand following the command-name. Examples of arguments
13 are given based on the following command line:

14
15 findstr /i /d:\winnt;\winnt\system32 aa*b *.ini.
16
17

18 In the above command line, "findstr" is argument 0, "/i" is argument 1,
19 "/d:\winnt;\winnt\system32" is argument 2, "aa*b" is argument 3, and "*.ini" is
20 argument 4. An "option" is an argument to a cmdlet that is generally used to
21 specify changes to the program's default behavior. Continuing with the example
22 command line above, "/i" and "/d" are options. An "option-argument" is an input
23 parameter that follows certain options. In some cases, an option-argument is
24 included within the same argument string as the option. In other cases, the option-
25 argument is included as the next argument. Referring again to the above

1 command line, "winnt;\winnt\system32" is an option-argument. An "operand" is
2 an argument to a cmdlet that is generally used as an object supplying information
3 to a program necessary to complete program processing. Operands generally
4 follow the options in a command line. Referring to the example command line
5 above again, "aa*b" and "*.ini" are operands. A "parsable stream" includes the
6 arguments.

7 Referring to FIGURE 13, parserparses a parsable stream (e.g., command
8 string 1350) into constituent parts 1320-1326 (e.g., where portion 1322). Each
9 portion 1320-1326 is associated with one of the cmdlets 1330-1336. Parserand
10 engineperform various processing, such as parsing, parameter validation, data
11 generation, parameter processing, parameter encoding, and parameter
12 documentation. Because parserand engineperform common functionality on the
13 input parameters on the command line, the administrative tool frameworkis able to
14 issue consistent error messages to users.

15 As one will recognize, the executable cmdlets 1330-1336 written in
16 accordance with the present administrative tool framework require less code than
17 commands in prior administrative environments. Each executable cmdlet 1330-
18 1336 is identified using its respective constituent part 1320-1326. In addition,
19 each executable cmdlet 1330-1336 outputs objects (represented by arrows 1340,
20 1342, 1344, and 1346) which are input as input objects (represented by arrows
21 1341, 1343, and 1345) to the next pipelined cmdlet. These objects may be input
22 by passing a reference (e.g., handle) to the object. The executable cmdlets 1330-
23 1336 may then perform additional processing on the objects that were passed in.
24
25

FIGURE 14 is a logical flow diagram illustrating in more detail the processing of command strings suitable for use within the process for handling input shown in FIGURE 9. The command string processing begins at block **1401**, where either the parser or the script engine identified a command string within the input. In general the core engine performs set-up and sequencing of the data flow of the cmdlets. The set-up and sequencing for one cmdlet is described below, but is applicable to each cmdlet in a pipeline. Processing continues at block **1404**.

At block **1404**, a cmdlet is identified. The identification of the cmdlet may be thru registration. The core engine determines whether the cmdlet is local or remote. The cmdlet may execute in the following locations: 1) within the application domain of the administrative tool framework; 2) within another application domain of the same process as the administrative tool framework; 3) within another process on the same computing device; or 4) within a remote computing device. The communication between cmdlets operating within the same process is through objects. The communication between cmdlets operating within different processes is through a serialized structured data format. One exemplary serialized structured data format is based on the extensible markup language (XML). Processing continues at block **1406**.

At block **1406**, an instance of the cmdlet object is created. An exemplary process for creating an instance of the cmdlet is described below in conjunction with FIGURE 15. Once the cmdlet object is created, processing continues at block **1408**.

At block **1408**, the properties associated with the cmdlet object are populated. As described above, the developer declares properties within a cmdlet class or within an external source. Briefly, the administrative tool framework will

1 decipher the incoming object(s) to the cmdlet instantiated from the cmdlet class
2 based on the name and type that is declared for the property. If the types are
3 different, the type may be coerced via the extended data type manager. As
4 mentioned earlier, in pipelined command strings, the output of each cmdlet may be
5 a list of handles to objects. The next cmdlet may inputs this list of object handles,
6 performs processing, and passes another list of object handles to the next cmdlet.
7 In addition, as illustrated in FIGURE 7, input parameters may be specified as
8 coming from the command line. One exemplary method for populating properties
9 associated with a cmdlet is described below in conjunction with FIGURE 16.
10 Once the cmdlet is populated, processing continues at block **1410**.

11 At block **1410**, the cmdlet is executed. In overview, the processing
12 provided by the cmdlet is performed at least once, which includes processing for
13 each input object to the cmdlet. Thus, if the cmdlet is the first cmdlet within a
14 pipelined command string, the processing is executed once. For subsequent
15 cmdlets, the processing is executed for each object that is passed to the cmdlet.
16 One exemplary method for executing cmdlets is described below in conjunction
17 with FIGURE 5. When the input parameters are only coming from the command
18 line, execution of the cmdlet uses the default methods provided by the base cmdlet
19 case. Once the cmdlet is finished executing, processing proceeds to block **1412**.

20 At block **1412**, the cmdlet is cleaned-up. This includes calling the
21 destructor for the associated cmdlet object which is responsible for de-allocating
22 memory and the like. The processing of the command string is then complete.

23 Exemplary Process for Creating a Cmdlet Object

24 FIGURE 15 is a logical flow diagram illustrating an exemplary process for
25 creating a cmdlet object suitable for use within the processing of command strings

1 shown in FIGURE 14. At this point, the cmdlet data structure has been developed
2 and attributes and expected input parameters have been specified. The cmdlet has
3 been compiled and has been registered. During registration, the class name (i.e.,
4 cmdlet name) is written in the registration store and the metadata associated with
5 the cmdlet has been stored. The process 1500 begins at block 1501, where the
6 parser has received input (e.g., keystrokes) indicating a cmdlet. The parser may
7 recognize the input as a cmdlet by looking up the input from within the registry
8 and associating the input with one of the registered cmdlets. Processing proceeds
9 to block 1504.

10 At block 1504, metadata associated with the cmdlet object class is read.
11 The metadata includes any of the directives associated with the cmdlet. The
12 directives may apply to the cmdlet itself or to one or more of the parameters.
13 During cmdlet registration, the registration code registers the metadata into a
14 persistent store. The metadata may be stored in an XML file in a serialized
15 format, an external database, and the like. Similar to the processing of directives
16 during script processing, each category of directives is processed at a different
17 stage. Each metadata directive handles its own error handling. Processing
18 continues at block 1506.

19 At block 1506, a cmdlet object is instantiated based on the identified cmdlet
20 class. Processing continues at block 1508.

21 At block 1508, information is obtained about the cmdlet. This may occur
22 through reflection or other means. The information is about the expected input
23 parameters. As mentioned above, the parameters that are declared public (e.g.,
24 public string Name 730) correspond to expected input parameters that can be
25

1 specified in a command string on a command line or provided in an input stream.
2 The administrative tool framework through the extended type manager, described
3 in FIGURE 18, provides a common interface for returning the information (on a
4 need basis) to the caller. Processing continues at block **1510**.

5 At block **1510**, applicability directives (e.g., Table 1) are applied. The
6 applicability directives insure that the class is used in certain machine roles and/or
7 user roles. For example, certain cmdlets may only be used by Domain
8 Administrators. If the constraint specified in one of the applicability directives is
9 not met, an error occurs. Processing continues at block **1512**.

10 At block **1512**, metadata is used to provide intellisense. At this point in
11 processing, the entire command string has not yet been entered. The
12 administrative tool framework, however, knows the available cmdlets. Once a
13 cmdlet has been determined, the administrative tool framework knows the input
14 parameters that are allowed by reflecting on the cmdlet object. Thus, the
15 administrative tool framework may auto-complete the cmdlet once a
16 disambiguating portion of the cmdlet name is provided, and then auto-complete
17 the input parameter once a disambiguating portion of the input parameter has been
18 typed on the command line. Auto-completion may occur as soon as the portion of
19 the input parameter can identify one of the input parameters unambiguously. In
20 addition, auto-completion may occur on cmdlet names and operands too.
21 Processing continues at block **1514**.

22 At block **1514**, the process waits until the input parameters for the cmdlet
23 have been entered. This may occur once the user has indicated the end of the
24 command string, such as by hitting a return key. In a script, a new line indicates
25

1 the end of the command string. This wait may include obtaining additional
2 information from the user regarding the parameters and applying other directives.
3 When the cmdlet is one of the pipelined parameters, processing may begin
4 immediately. Once, the necessary command string and input parameters have
5 been provided, processing is complete.

6 Exemplary Process for Populating the Cmdlet

7 An exemplary process for populating a cmdlet is illustrated in FIGURE 16
8 and is now described, in conjunction with FIGURE 5. In one exemplary
9 administrative tool framework, the core engine performs the processing to
10 populate the parameters for the cmdlet. Processing begins at block 1601 after an
11 instance of a cmdlet has been created. Processing continues to block 1602.

12 At block 1602, a parameter (e.g., ProcessName) declared within the cmdlet
13 is retrieved. Based on the declaration with the cmdlet, the core engine recognizes
14 that the incoming input objects will provide a property named "ProcessName". If
15 the type of the incoming property is different than the type specified in the
16 parameter declaration, the type will be coerced via the extended type manager.
17 The process of coercing data types is explained below in the subsection entitled
18 "Exemplary Extended Type Manager Processing." Processing continues to block
19 1603.

20 At block 1603, an attribute associated with the parameter is obtained. The
21 attribute identifies whether the input source for the parameter is the command line
22 or whether it is from the pipeline. Processing continues to decision block 1604.

23 At decision block 1604, a determination is made whether the attribute
24 specifies the input source as the command line. If the input source is the
25

1 command line, processing continues at block 1609. Otherwise, processing
2 continues at decision block 1605.

3 At decision block **1605**, a determination is made whether the property name
4 specified in the declaration should be used or whether a mapping for the property
5 name should be used. This determination is based on whether the command input
6 specified a mapping for the parameter. The following line illustrates an exemplary
7 mapping of the parameter “ProcessName” to the “foo” member of the incoming
8 object:

9 \$ get/process | where han* -gt 500 | stop/process -ProcessName<-foo.

10 Processing continues at block **1606**.

11 At block **1606**, the mapping is applied. The mapping replaces the name of
12 the expected parameter from “ProcessName” to “foo”, which is then used by the
13 core engine to parse the incoming objects and to identify the correct expected
14 parameter. Processing continues at block **1608**.

15 At block **1608**, the extended type manager is queried to locate a value for
16 the parameter within the incoming object. As explain in conjunction with the
17 extended type manager, the extended type manager takes the parameter name and
18 uses reflection to identify a parameter within the incoming object with parameter
19 name. The extended type manager may also perform other processing for the
20 parameter, if necessary. For example, the extended type manager may coerce the
21 type of data to the expected type of data through a conversion mechanism
22 described above. Processing continues to decision block **1610**.

23 Referring back to block 1609, if the attribute specifies that the input source
24 is the command line, data from the command line is obtained. Obtaining the data
25

1 from the command line may be performed via the extended type manager.
2 Processing then continues to decision block 1610.

3 At decision block 1610, a determination is made whether there is another
4 expected parameter. If there is another expected parameter, processing loops back
5 to block 1602 and proceeds as described above. Otherwise, processing is
6 complete and returns.

7 Thus, as shown, cmdlets act as a template for shredding incoming data to
8 obtain the expected parameters. In addition, the expected parameters are obtained
9 without knowing the type of incoming object providing the value for the expected
10 parameter. This is quite different than traditional administrative environments.
11 Traditional administrative environments are tightly bound and require that the type
12 of object be known at compile time. In addition, in traditional environments, the
13 expected parameter would have been passed into the function by value or by
14 reference. Thus, the present parsing (e.g., "shredding") mechanism allows
15 programmers to specify the type of parameter without requiring them to
16 specifically know how the values for these parameters are obtained.

17 For example, given the following declaration for the cmdlet Foo:

```
18  
19 class Foo : Cmdlet  
20  
21 {  
22     string Name;  
23     Bool Recurse;  
24 }  
25
```

1
2 The command line syntax may be any of the following:
3

4 \$ Foo -Name: (string) -Recurse: True
5

6 \$ Foo -Name <string> -Recurse True
7

8 \$Foo -Name (string).
9

10 The set of rules may be modified by system administrators in order to yield
11 a desired syntax. In addition, the parser may support multiple sets of rules, so that
12 more than one syntax can be used by users. In essence, the grammar associated
13 with the cmdlet structure (e.g., string Name and Bool Recurse) drives the parser.

14 In general, the parsing directives describe how the parameters entered as
15 the command string should map to the expected parameters identified in the
16 cmdlet object. The input parameter types are checked to determine whether
17 correct. If the input parameter types are not correct, the input parameters may be
18 coerced to become correct. If the input parameter types are not correct and can not
19 be coerced, a usage error is printed. The usage error allows the user to become
20 aware of the correct syntax that is expected. The usage error may obtain
21 information describing the syntax from the Documentation Directives. Once the
22 input parameter types have either been mapped or have been verified, the
23 corresponding members in the cmdlet object instance are populated. As the
24 members are populated, the extended type manager provides processing of the
25 input parameter types. Briefly, the processing may include a property path

1 mechanism, a key mechanism, a compare mechanism, a conversion mechanism, a
2 globber mechanism, a relationship mechanism, and a property set mechanism.
3 Each of these mechanisms is described in detail below in the section entitled
4 “Extended Type Manager Processing”, which also includes illustrative examples.

5 Exemplary Process for Executing the Cmdlet

6 An exemplary process for executing a cmdlet is illustrated in FIGURE 17
7 and is now described. In one exemplary administrative tool environment, the core
8 engine executes the cmdlet. As mentioned above, the code **1442** within the second
9 method **1440** is executed for each input object. Processing begins at block **1701**
10 where the cmdlet has already been populated. Processing continues at block **1702**.

11 At block **1702**, a statement from the code **542** is retrieved for execution.
12 Processing continues at decision block **1704**.

13 At decision block **1704**, a determination is made whether a hook is included
14 within the statement. Turning briefly to FIGURE 5, the hook may include calling
15 an API provided by the core engine. For example, statement **550** within the code
16 **542** of cmdlet **500** in FIGURE 5 calls the confirmprocessing API specifying the
17 necessary parameters, a first string (e.g., “PID=”), and a parameter (e.g., PID).
18 Turning back to FIGURE 17, if the statement includes the hook, processing
19 continues to block **1712**. Thus, if the instruction calling the confirmprocessing
20 API is specified, the cmdlet operates in an alternate executing mode that is
21 provided by the operating environment. Otherwise, processing continues at block
22 **1706** and execution continues in the “normal” mode.

23 At block **1706**, the statement is processed. Processing then proceeds to
24 decision block **1708**. At block **1708**, a determination is made whether the code
25

1 includes another statement. If there is another statement, processing loops back to
2 block **1702** to get the next statement and proceeds as described above. Otherwise,
3 processing continues to decision block **1714**.

4 At decision block **1714**, a determination is made whether there is another
5 input object to process. If there is another input object, processing continues to
6 block **1716** where the cmdlet is populated with data from the next object. The
7 population process described in FIGURE 16 is performed with the next object.
8 Processing then loops back to block **1702** and proceeds as described above. Once
9 all the objects have been processed, the process for executing the cmdlet is
10 complete and returns.

11 Returning back to decision block **1704**, if the statement includes the hook,
12 processing continues to block **1712**. At block **1712**, the additional features
13 provided by the administrative tool environment are processed. Processing
14 continues at decision block **1708** and continues as described above.

15 The additional processing performed within block **1712** is now described in
16 conjunction with the exemplary data structure **600** illustrated in FIGURE 6. As
17 explained above, within the command base class **600** there may be parameters
18 declared that correspond to additional expected input parameters (e.g., a switch).

19 The switch includes a predetermined string, and when recognized, directs
20 the core engine to provide additional functionality to the cmdlet. If the parameter
21 verbose **610** is specified in the command input, verbose statements **614** are
22 executed. The following is an example of a command line that includes the
23 verbose switch:

24
25 \$ get/process | where "han* -gt 500" | stop/process -verbose.

1
2 In general, when “-verbose” is specified within the command input, the
3 core engine executes the command for each input object and forwards the actual
4 command that was executed for each input object to the host program for display.
5 The following is an example of output generated when the above command line is
6 executed in the exemplary administrative tool environment:

7
8 \$ stop/process PID=15

9 \$ stop/process PID=33.
10

11 If the parameter whatif **620** is specified in the command input, whatif
12 statements **624** are executed. The following is an example of a command line that
13 includes the whatif switch:

14
15 \$ get/process | where “han* -gt 500” | stop/process –whatif.
16

17 In general, when “-whatif” is specified, the core engine does not actually
18 execute the code **542**, but rather sends the commands that would have been
19 executed to the host program for display. The following is an example of output
20 generated when the above command line is executed in the administrative tool
21 environment of the present invention:

22
23 # \$ stop/process PID=15

24 # \$ stop/process PID=33.
25

1 If the parameter confirm **630** is specified in the command input, confirm
2 statements **634** are executed. The following is an example of a command line that
3 includes the confirm switch:

4
5 \$ get/process | where "han* -gt 500" | stop/process -confirm.

6
7 In general, when "-confirm" is specified, the core engine requests
8 additional user input on whether to proceed with the command or not. The
9 following is an example of output generated when the above command line is
10 executed in the administrative tool environment of the present invention.

11
12 \$ stop/process PID 15

13 Y/N Y

14 \$ stop/process PID 33

15 Y/N N.

16
17 As described above, the exemplary data structure **600** may also include a
18 security method **640** that determines whether the task being requested for
19 execution should be allowed. In traditional administrative environments, each
20 command is responsible for checking whether the person executing the command
21 has sufficient privileges to perform the command. In order to perform this check,
22 extensive code is needed to access information from several sources. Because of
23 these complexities, many commands did not perform a security check. The
24 inventors of the present administrative tool environment recognized that when the
25 task is specified in the command input, the necessary information for performing

1 the security check is available within the administrative tool environment.
2 Therefore, the administrative tool framework performs the security check without
3 requiring complex code from the tool developers. The security check may be
4 performed for any cmdlet that defines the hook within its cmdlet. Alternatively,
5 the hook may be an optional input parameter that can be specified in the command
6 input, similar to the verbose parameter described above.

7 The security check is implemented to support roles based authentication,
8 which is generally defined as a system of controlling which users have access to
9 resources based on the role of the user. Thus, each role is assigned certain access
10 rights to different resources. A user is then assigned to one or more roles. In
11 general, roles based authentication focus on three items: principle, resource, and
12 action. The *principle* identifies who requested the *action* to be performed on the
13 *resource*.

14 The inventors of the present invention recognized that the cmdlet being
15 requested corresponded to the action that was to be performed. In addition, the
16 inventors appreciated that the owner of the process in which the administrative
17 tool framework was executing corresponded to the principle. Further, the
18 inventors appreciated that the resource is specified within the cmdlet. Therefore,
19 because the administrative tool framework has access to these items, the inventors
20 recognized that the security check could be performed from within the
21 administrative tool framework without requiring tool developers to implement the
22 security check.

23 The operation of the security check may be performed any time additional
24 functionality is requested within the cmdlet by using the hook, such as the
25 confirmprocessing API. Alternatively, security check may be performed by

1 checking whether a security switch was entered on the command line, similar to
2 verbose, whatif, and confirm. For either implementation, the checkSecurity
3 method calls an API provided by a security process (not shown) that provides a set
4 of APIs for determining who is allowed. The security process takes the
5 information provided by the administrative tool framework and provides a result
6 indicating whether the task may be completed. The administrative tool framework
7 may then provide an error or just stop the execution of the task.

8 Thus, by providing the hook within the cmdlet, the developers may use
9 additional processing provided by the administrative tool framework.

10 Exemplary Extended Type Manager Processing

11 As briefly mentioned above in conjunction with FIGURE 18, the extended
12 type manager may perform additional processing on objects that are supplied. The
13 additional processing may be performed at the request of the parser 220, the script
14 engine 222, or the pipeline processor 402. The additional processing includes a
15 property path mechanism, a key mechanism, a compare mechanism, a conversion
16 mechanism, a globber mechanism, a relationship mechanism, and a property set
17 mechanism. Those skilled in the art will appreciate that the extended type
18 manager may also be extended with other processing without departing from the
19 scope of the claimed invention. Each of the additional processing mechanisms is
20 now described.

21 First, the property path mechanism allows a string to navigate properties of
22 objects. In current reflection systems, queries may query properties of an object.
23 However, in the present extended type manager, a string may be specified that will
24 provide a navigation path to successive properties of objects. The following is an
25 illustrative syntax for the property path: P1.P2.P3.P4.

Each component (e.g., P1, P2, P3, and P4) comprises a string that may represent a property, a method with parameters, a method without parameters, a field, an XPATH, or the like. An XPATH specifies a query string to search for an element (e.g., “/FOO@=13”). Within the string, a special character may be included to specifically indicate the type of component. If the string does not contain the special character, the extended type manager may perform a lookup to determine the type of component. For example, if component P1 is an object, the extended type manager may query whether P2 is a property of the object, a method on the object, a field of the object, or a property set. Once the extended type manager identifies the type for P2, processing according to that type is performed. If the component is not one of the above types, the extended type manager may further query the extended sources to determine whether there is a conversion function to convert the type of P1 into the type of P2. These and other lookups will now be described using illustrative command strings and showing the respective output.

The following is an illustrative string that includes a property path:

```
$ get/process | /where hand* -gt> 500 | format/table name.toupper, ws.kb,  
exe*.ver*.description.tolower.trunc(30).
```

In the above illustrative string, there are three property paths: (1) “name.toupper”; (2) “ws.kb”; and (3) “exe*.ver*.description.tolower.trunc(30). Before describing these property paths, one should note that “name”, “ws”, and “exe” specify the properties for the table. In addition, one should note that each of these properties is a direct property of the incoming object, originally generated by

1 “get/process” and then pipelined through the various cmdlets. Processing
2 involved for each of the three property paths will now be described.

3 In the first property path (i.e., “name.toupper”), name is a direct property of
4 the incoming object and is also an object itself. The extended type manager
5 queries the system using the priority lookup described above to determine the type
6 for toupper. The extended type manager discovers that toupper is not a property.
7 However, toupper may be a method inherited by a string type to convert lower
8 case letters to upper case letters within the string. Alternatively, the extended type
9 manager may have queried the extended metadata to determine whether there is
10 any third party code that can convert a name object to upper case. Upon finding
11 the component type, processing is performed in accordance with that component
12 type.

13 In the second property path (i.e., “ws.kb”), “ws” is a direct property of the
14 incoming object and is also an object itself. The extended type manager
15 determines that “ws” is an integer. Then, the extended type manager queries
16 whether kb is a property of an integer, whether kb is a method of an integer, and
17 finally queries whether any code knows how to take an integer and convert the
18 integer to a kb type. Third party code is registered to perform this conversion and
19 the conversion is performed.

20 In the third property path (i.e., “exe*.ver*.description.tolower.trunc(30)”)
21 there are several components. The first component (“exe*”) is a direct property of
22 the incoming object and is also an object. Again, the extended type manager
23 proceeds down the lookup query in order to process the second component
24 (“ver*”). The “exe*” object does not have a “ver*” property or method, so the
25

1 extend type manager queries the extended metadata to determine whether there is
2 any code that is registered to convert an executable name into a version. For this
3 example, such code exists. The code may take the executable name string and use
4 it to open a file, then accesses the version block object, and return the description
5 property (the third component ("description") of the version block object. The
6 extended type manager then performs this same lookup mechanism for the fourth
7 component ("tolower") and the fifth component ("trunc(40)"). Thus, as
8 illustrated, the extended type manager may perform quite elaborate processing on
9 a command string without the administrator needing to write any specific code.

10 Table 1 illustrates output generated for the illustrative string.

11
12 Name.toupper ws.kb exe*.ver*.description.tolower.trunc(30)

13 ETCLIENT 29,964 etclient

14 CSRSS 6,944

15 SVCHOST 28,944 generic host process for win32

16 OUTLOOK 18,556 office outlook

17 MSMSGs 13,248 messenger

18 Table 1.

19 Another query mechanism 1824 includes a key. The key identifies one or
20 more properties that make an instance of the data type unique. For example, in a
21 database, one column may be identified as the key which can uniquely identify
22 each row (e.g., social security number). The key is stored within the type
23 metadata 1840 associated with the data type. This key may then be used by the
24 extended type manager when processing objects of that data type. The data type
25 may be an extended data type or an existing data type.

Another query mechanism 1824 includes a compare mechanism. The compare mechanism compares two objects. If the two objects directly support the compare function, the directly supported compare function is executed. However, if neither object supports a compare function, the extended type manager may look in the type metadata for code that has been registered to support the compare between the two objects. An illustrative series of command line strings invoking the compare mechanism is shown below, along with corresponding output in Table 2.

```
$ $a = $( get/date )
$ start/sleep 5
$ $b = $( get/date
compare/time $a $b

Ticks      : 51196579
Days       : 0
Hours      : 0
Milliseconds : 119
Minutes    : 0
Seconds    : 5
TotalDays  : 5.92552997685185E-05
TotalHours : 0.00142212719444444
TotalMilliseconds : 5119.6579
TotalMinutes : 0.0853276316666667
TotalSeconds : 5.1196579
```

Table 2.

1 Compare/time cmdlet is written to compare two datetime objects. In this
2 case, the DateTime object supports the IComparable interface.

3
4 Another query mechanism 1824 includes a conversion mechanism. The
5 extended type manager allows code to be registered stating its ability to perform a
6 specific conversion. Then, when an object of type A is input and a cmdlet
7 specifies an object of type B, the extended type manager may perform the
8 conversion using one of the registered conversions. The extended type manager
9 may perform a series of conversions in order to coerce type A into type B. The
10 property path described above ("ws.kb") illustrates a conversion mechanism.

11
12 Another query mechanism 1824 includes a globber mechanism. A globber
13 refers to a wild card character within a string. The globber mechanism inputs the
14 string with the wild card character and produces a set of objects. The extended
15 type manager allows code to be registered that specifies wildcard processing. The
16 property path described above ("exe*.ver*.description.tolower.trunc(30))
17 illustrates the globber mechanism. A registered process may provide globbing for
18 file names, file objects, incoming properties, and the like.

19 Another query mechanism 1824 includes a property set mechanism. The
20 property set mechanism allows a name to be defined for a set of properties. An
21 administrator may then specify the name within the command string to obtain the
22 set of properties. The property set may be defined in various ways. In one way, a
23 predefined parameter, such as "?", may be entered as an input parameter for a
24 cmdlet. The operating environment upon recognizing the predefined parameter
25 lists all the properties of the incoming object. The list may be a GUI that allows

an administrator to easily check (e.g., “click on”) the properties desired and name the property set. The property set information is then stored in the extended metadata. An illustrative string invoking the property set mechanism is shown below, along with corresponding output in Table 3:

```
$ get/process | where han* -gt 500 | format/table config.
```

In this illustrative string, a property set named “config” has been defined to include a name property, a process id property (Pid), and a priority property. The output for the table is shown below.

<u>Name</u>	<u>Pid</u>	<u>Priority</u>
ETClient	3528	Normal
csrss	528	Normal
svchost	848	Normal
OUTLOOK	2,772	Normal
msmsgs	2,584	Normal

Table 3.

Another query mechanism 1824 includes a relationship mechanism. In contrast to traditional type systems that support one relationship (i.e., inheritance), the relationship mechanism supports expressing more than one relationship between types. Again, these relationships are registered. The relationship may include finding items that the object consumes or finding the items that consume the object. The extended type manager may access ontologies that describe various relationships. Using the extended metadata and the code, a specification for accessing any ontology service, such as OWL, DAWL, and the like, may be

described. The following is a portion of an illustrative string which utilizes the relationship mechanism: .OWL:"string".

The "OWL" identifier identifies the ontology service and the "string" specifies the specific string within the ontology service. Thus, the extended type manager may access types supplied by ontology services.

Exemplary Process for Displaying Command Line Data

The present mechanism provides a data driven command line output. The formatting and outputting of the data is provided by one or more cmdlets in the pipeline of cmdlets. Typically, these cmdlets are included within the non-management cmdlets described in conjunction with FIGURE 2 above. The cmdlets may include a format cmdlet, a markup cmdlet, a convert cmdlet, a transform cmdlet, and an out cmdlet.

FIGURE 19 graphically depicts exemplary sequences 1901-1907 of these cmdlets within a pipeline. The first sequence 1901 illustrates the out cmdlet 1910 as the last cmdlet in the pipeline. In the same manner as described above for other cmdlets, the out cmdlet 1910 accepts a stream of pipeline objects generated and processed by other cmdlets within the pipeline. However, in contrast to most cmdlets, the out cmdlet 1910 does not emit pipeline objects for other cmdlets. Instead, the out cmdlet 1910 is responsible for rendering/displaying the results generated by the pipeline. Each out cmdlet 1910 is associated with an output destination, such as a device, a program, and the like. For example, for a console device, the out cmdlet 1910 may be specified as out/console; for an internet browser, the out cmdlet 1910 may be specified as out/browser; and for a window, the out cmdlet 1910 may be specified as out/window. Each specific out cmdlet is

1 familiar with the capabilities of its associated destination. Locale information
2 (e.g., date & currency formats) are processed by the out cmdlet 1910, unless a
3 convert cmdlet preceded the out cmdlet in the pipeline. In this situation, the
4 convert cmdlet processed the local information.

5 Each host is responsible for supporting certain out cmdlets, such as
6 out/console. The host also supports any destination specific host cmdlet (e.g.,
7 out/chart that directs output to a chart provided by a spreadsheet application). In
8 addition, the host is responsible for providing default handling of results. The out
9 cmdlet in this sequence may decide to implement its behavior by calling other
10 output processing cmdlets (such as format/markup/convert/transform). Thus, the
11 out cmdlet may implicitly modify sequence 1901 to any of the other sequences or
12 may add its own additional format/output cmdlets.

13 The second sequence 1902 illustrates a format cmdlet 1920 before the out
14 cmdlet 1910. For this sequence, the format cmdlet 1920 accepts a stream of
15 pipeline objects generated and processed by other cmdlets within the pipeline. In
16 overview, the format cmdlet 1920 provides a way to select display properties and a
17 way to specify a page layout, such as shape, column widths, headers, footers, and
18 the like. The shape may include a table, a wide list, a columnar list, and the like.
19 In addition, the format cmdlet 1920 may include computations of totals or sums.
20 Exemplary processing performed by a format cmdlet 1920 is described below in
21 conjunction with FIGURE 20. Briefly, the format cmdlet emits format objects, in
22 addition to emitting pipeline objects. The format objects can be recognized
23 downstream by an out cmdlet (e.g., out cmdlet 1920 in sequence 1902) via the
24 extended type manager or other mechanism. The out cmdlet 1920 may choose to
25 either use the emitted format objects or may choose to ignore them. The out

1 cmdlet determines the page layout based on the page layout data specified in the
2 display information. In certain instances, modifications to the page layout may be
3 specified by the out cmdlet. In one exemplary process the out cmdlet may
4 determine an unspecified column width by finding a maximum length for each
5 property of a predetermined number of objects (e.g., 50) and setting the column
6 width to the maximum length. The format objects include formatting information,
7 header/footer information, and the like.

8 The third sequence 1903 illustrates a format cmdlet 1920 before the out
9 cmdlet 1910. However, in the third sequence 1903, a markup cmdlet 1930 is
10 pipelined between the format cmdlet 1920 and the out cmdlet 1910. The markup
11 cmdlet 1930 provides a mechanism for adding property annotation (e.g., font,
12 color) to selected parameters. Thus, the markup cmdlet 1930 appears before the
13 output cmdlet 1910. The property annotations may be implemented using a
14 "shadow property bag", or by adding property annotations in a custom namespace
15 in a property bag. The markup cmdlet 1930 may appear before the format cmdlet
16 1920 as long as the markup annotations may be maintained during processing of
17 the format cmdlet 1920.

18 The fourth sequence 1904 again illustrates a format cmdlet 1920 before the
19 out cmdlet 1910. However, in the fourth sequence 1904, a convert cmdlet 1940 is
20 pipelined between the format cmdlet 1920 and the out cmdlet 1910. The convert
21 cmdlet 1940 is also configured to process the format objects emitted by the format
22 cmdlet 1920. The convert cmdlet 1940 converts the pipelined objects into a
23 specific encoding based on the format objects. The convert cmdlet 1940 is
24 associated with the specific encoding. For example, the convert cmdlet 1940 that
25 converts the pipelined objects into Active Directory Objects (ADO) may be

1 declared as “convert/ADO” on the command line. Likewise, the convert cmdlet
2 1940 that converts the pipelined objects into comma separated values (csv) may be
3 declared as “convert/csv” on the command line. Some of the convert cmdlets
4 1940 (e.g., convert/XML and convert/html) may be blocking commands, meaning
5 that all the pipelined objects are received before executing the conversion.
6 Typically, the out cmdlet 1920 may determine whether to use the formatting
7 information provided by the format objects. However, when a convert cmdlet
8 1920 appears before the out cmdlet 1920, the actual data conversion has already
9 occurred before the out cmdlet receives the objects. Therefore, in this situation,
10 the out cmdlet can not ignore the conversion.

11 The fifth sequence 1905 illustrates a format cmdlet 1920, a markup cmdlet
12 1930, a convert cmdlet 1940, and an out cmdlet 1910 in that order. Thus, this
13 illustrates that the markup cmdlet 1930 may occur before the convert cmdlet 1940.

14 The sixth sequence 1906 illustrates a format cmdlet 1920, a specific convert
15 cmdlet (e.g., convert/xml cmdlet 1940’), a specific transform cmdlet (e.g.,
16 transform/xslt cmdlet 1950), and an out cmdlet 1910. The convert/xml cmdlet
17 1940’ converts the pipelined objects into an extended markup language (XML)
18 document. The transform/xslt cmdlet 1950 transforms the XML document into
19 another XML document using an Extensible Style Language (XSL) style sheet. The
20 transform process is commonly referred to as extensible style language
21 transformation (XSLT), in which an XSL processor reads the XML document and
22 follows the instructions within the XSL style sheet to create the new XML
23 document.

24 The seventh sequence 1907 illustrates a format cmdlet 1920, a markup
25 cmdlet 1930, a specific convert cmdlet (e.g., convert/xml cmdlet 1940’), a specific

1 transform cmdlet (e.g., transform/xslt cmdlet 1950), and an out cmdlet 1910.
2 Thus, the seventh sequence 1907 illustrates having the markup cmdlet 1930
3 upstream from the convert cmdlet and transform cmdlet.

4 FIGURE 20 illustrates exemplary processing 2000 performed by a format
5 cmdlet. The formatting process begins at block 2001, after the format cmdlet has
6 been parsed and invoked by the parser and pipeline processor in a manner
7 described above. Processing continues at block 2002.

8 At block 2002, a pipeline object is received as input to the format cmdlet.
9 Processing continues at block 2004.

10 At block 2004, a query is initiated to identify a type for the pipelined
11 object. This query is performed by the extended type manager as described above
12 in conjunction with FIGURE 18. Once the extended type manager has identified
13 the type for the object, processing continues at block 2006.

14 At block 2006, the identified type is looked up in display information. An
15 exemplary format for the display information is illustrated in FIGURE 21 and will
16 be described below. Processing continues at decision block 2008.

17 At decision block 2008, a determination is made whether the identified type
18 is specified within the display information. If there is no entry within the display
19 information for the identified type, processing is complete. Otherwise, processing
20 continues at block 2010.

21 At block 2010, formatting information associated with the identified type is
22 obtained from the display information. Processing continues at block 2012.

23 At block 2012, information is emitted on the pipeline. Once the
24 information is emitted, the processing is complete.
25

1 Exemplary information that may be emitted is now described in further
2 detail. The information may include formatting information, header/footer
3 information, and a group end/begin signal object. The formatting information may
4 include a shape, a label, numbering/bullets, column widths, character encoding
5 type, content font properties, page length, group-by-property name, and the like.
6 Each of these may have additional specifications associated with it. For example,
7 the shape may specify whether the shape is a table, a list, or the like. Labels may
8 specify whether to use column headers, list labels, or the like. Character encoding
9 may specify ASCII, UTF-8, Unicode, and the like. Content font properties may
10 specify the font that is applied to the property values that are display. A default
11 font property (e.g., Courier New, 10 point) may be used if content font properties
12 are not specified.

13 The header/footer information may include a header/footer scope, font
14 properties, title, subtitle, date, time, page numbering, separator, and the like. For
15 example, the scope may specify a document, a page, a group, or the like.
16 Additional properties may be specified for either the header or the footer. For
17 example, for group and document footers, the additional properties may include
18 properties or columns to calculate a sum/total, object counts, label strings for totals
19 and counts, and the like.

20 The group end/begin signal objects are emitted when the format cmdlet
21 detects that a group-by property has changed. When this occurs, the format cmdlet
22 treats the stream of pipeline objects as previously sorted and does not re-sort them.
23 The group end/begin signal objects may be interspersed with the pipeline objects.
24 Multiple group-by properties may be specified for nested sorting. The format
25 cmdlet may also emit a format end object that includes final sums and totals.

Turning briefly to FIGURE 21, an exemplary display information 2100 is in a structured format and contains information (e.g., formatting information, header/footer information, group-by properties or methods) associated with each object that has been defined. For example, the display information 2100 may be XML-based. Each of the afore-mentioned properties may then be specified within the display information. The information within the display information 2100 may be populated by the owner of the object type that is being entered. The operating environment provides certain APIs and cmdlets that allow the owner to update the display information by creating, deleting, and modifying entries.

FIGURE 22 is a table listing an exemplary syntax 2201-2213 for certain format cmdlets (e., format/table, format/list, and format/wide), markup cmdlets (e.g., add/markup), convert cmdlets (e.g., convert/text, convert/sv, convert/csv, convert/ADO, convert/XML, convert/html), transform cmdlets (e.g., transform/XSLT) and out cmdlets (e.g., out/console, out/file). FIGURE 23 illustrates results rendered by the out/console cmdlet using various pipeline sequences of the output processing cmdlets (e.g., format cmdlets, convert cmdlets, and markup cmdlets).

As described, the mechanism for extending data types by analyzing partially unresolved input may be employed in an administrative tool environment. However, those skilled in the art will appreciate that the mechanism may be employed in various environments that need to specify and operate on various input.

Although details of specific implementations and embodiments are described above, such details are intended to satisfy statutory disclosure obligations rather than to limit the scope of the following claims. Thus, the

1 invention as defined by the claims is not limited to the specific features described
2 above. Rather, the invention is claimed in any of its forms or modifications that
3 fall within the proper scope of the appended claims, appropriately interpreted in
4 accordance with the doctrine of equivalents.
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25